Advanced Control Systems Lab

ACSL TurtleBot3 e-Manual

June 2020

Contents

Abstract

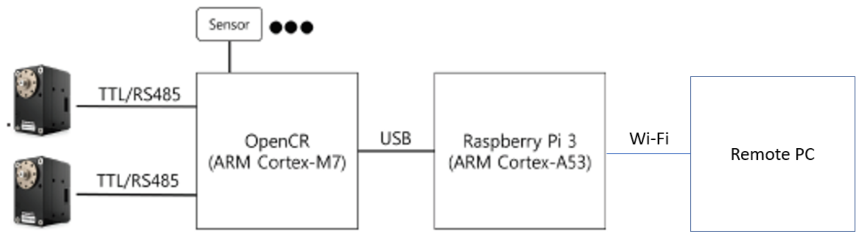
This guide contains the documentation for the customized `TurtleBot` developed for Advanced Control Systems Laboratory. The purpose of this Summer 2020 project was to deliver a modular reconfigurable PID controller for the Turtlebot wheel actuators to further the research of ACSL on SLAM, Navigation and Control Algorithms. The entire software has been written using ROS (`roscpp` and `rospy` libraries) and is present on 3 different machines. The software is based on the original code written by ROBOTIS for a stock `Turtlebot3`. The `Turtlebot3 eManuel` for ROS1 is the best starting point to get an initial understanding of the software.

This guide is not meant to be a comprehensive tutorial for ROS, Ubuntu Shell, Git or any tools used for development like Arduino IDE, Visual Studio Code etc. Understanding of these tools is assumed to be a pre-requisite, and instructions on how to "Bring Up" a new `TurtleBot3` have been provided. There exists a detailed description of the customizations done to the stock code, without repetitions from external literature. For content outside the guide, links have been provided at the appropriate places for further reading. However, this guide should be enough to get a new `TurtleBot3` up and running with ACSL Software.

1 Overview and Setup

TurtleBot3 is a small ROS-based mobile robot. The TurtleBot3 can be customized by changing the source code and addition of new hardware. The TurtleBot3 model used in this project is Waffle Pi. It is equipped with a Raspberry Pi Camera Module (v2), a 360-degree Laser Distance Sensor (LDS) and Dynamixel XM430 210-T motors.

`TurtleBot3` architecture is described in the following image.



The four major components of `TurtleBot3` are

1. Raspberry Pi 3B+ Single Board Computer (SBC).
2. OpenCR Embedded System Board.
3. LDS and Camera Sensors.
4. Dynamixel Actuators.

Raspberry Pi has a WiFi module built onto the board which connects to a computer, known as the Remote PC. Remote PC is used to send goal waypoints for SLAM and Navigation to TB3. Visualization of the robot is also performed on the remote PC using a ROS-based tool called `rviz`.
There is a one time setup that needs to be performed on the Remote PC, Raspberry Pi and OpenCR board. The following sections detail the procedure to setup the software on various components.

1.1 Remote PC Setup

To begin you will need to install the appropriate ROS version on your remote PC. The remote PC will be running ROS Melodic (EOL date: May 2023).

1.1.1 Quick Installation using Shell Script (recommended)

A shell script `install_melodic.sh` has been created to ease the installation of ROS and dependent packages on the Remote PC. This is located on the GitHub repo and can be cloned from there. Open a terminal and run the following commands.

```
git clone https://github.com/hanyiabc/ACSL_turtlebot3.git
cd ASCL_turtlebot3/
chmod 755 ./install_melodic.sh && bash ./install_melodic.sh
```

If you face any errors that cannot be resolved, the `install_melodic.sh` file is readable and you can execute those commands individually.

1.1.2 Manual Installation of ROS Melodic

If the shell script installation doesn't succeed, you can also follow instructions on ROS Wiki for setting up ROS Melodic on Ubuntu 18.04 manually. Complete the instructions on the following website.

`http://wiki.ros.org/melodic/Installation/Ubuntu`

Then, install required dependencies by running the last 3 commands in the `install_melodic.sh` file. They can be easily found under the "Install Dependencies" section.
Finally clone the repository using the following commands.

```
git clone https://github.com/hanyiabc/ACSL_turtlebot3.git
```

1.1.3 Network Configuration

Steps for configuring the wifi on the remote pc are taken from the following site and requirement specific instructions have been mentioned below.

`https://emanual.robotis.com/docs/en/platform/turtlebot3/pc_setup/#network-configuration`

For this project we will use the Linksys Router on the **Linksys04294** network. It is helpful to write down the IP address of the remote pc, since you will need it later to setup the turtlebot pc network settings.

1.1.4 Compiling Source Code on Remote PC

Once the network is setup on the Remote PC, go ahead and build the GitHub repository. This can be done by navigating to the top level folder of the repository `ASCL_turtlebot3/` and running `catkin_make` in the terminal.

Finally add the environment variable for the `TurtleBot3` model to `.bashrc` file and source it. Possible values are *waffle, waffle_pi, burger*. We have our own customized model and hence we have defined our own model name. Run the following commands to add the variables to the `.bashrc` file.

```
echo "export␣TURTLEBOT3_MODEL=waffle_pi_effort_controller" >> ~/.bashrc
source ~/.bashrc
```

A different URDF file (an XML file for the ROS robot model) is loaded based on this environment variable. For this project however, we have defined new URDF file for effort control. By adding the above environment variable, ROS would use the appropriate file to model the robot for visualization and take the appropriate geometrical dimensions/parameters in the code.

This completes the Remote PC setup. It is advised to open `.bashrc` file using `Gedit` or any other text editor to make sure there are no duplicate commands being run. Having duplicates doesn't lead to any errors, but some variables might get redefined and it is not considered good practice to have multiple commands for setting up the same environment variables or aliases.

1.2 Rasberry Pi SBC Setup

The single board computer on `TurtleBot3` will be running ROS Kinetic (EOL date: April 2021) for the ease of setup since ROS Melodic requires manual compilation when using Rasbian. No problem has yet been found by us on the communication between these 2 releases. Now it is time to look at the turtlebot SBC. For the turtlebot 3 Waffle Pi model, the on board PC is a raspberry PI. This step requires a micro SD card with adapter, a monitor with an HDMI input, a USB keyboard and mouse, and a power source for the turtlebot. ROBOTIS provides a prebuilt desktop environment for the turtlebot with ROS kinetic. Instructions for flashing this distribution can be found here (**Step 6.2.1.2 Install Linux Based on Raspbian, Do not use the other two methods.**).
`https://emanual.robotis.com/docs/en/platform/turtlebot3/raspberry_pi_3_setup/#raspberry-pi-3-setup`
To use the Raspberry PI without a physical access, use SSH. Example SSH command:

```
ssh pi@YOUR_RASPBERRY_PI_IP
```

For the ease of use, VNC configuration is recommended. This provide remote desktop funtionality. In case of Debian Stretch (the version that the manual will use), install Real VNC by running this command on the Raspberry PI:

```
sudo apt update
sudo apt install realvnc-vnc-server realvnc-vnc-viewer
```

THen enable VNC server by

```
sudo raspi-config
```

Then navigate to Interfacing Options. Scroll down and select VNC - Yes.

Ubuntu Desktop 18.04 comes with VNC client Remmina. On Windows, use VNC viewer `https://www.realvnc.com/en/connect/download/viewer/`

1.3 OpenCR Setup

Most of the software is located in the form of a binary file that is burnt to the EEPROM of the STM32F746 chip on the OpenCR board. As the software used for ACSL projects is custom, it is burnt using Arduino IDE.

The original source code for `TurtleBot3` used the inbuilt PID controller of the Dynamixel XM430 210-T actuators. However, to benchmark the performance of the controllers, the control is shifted from the motors to higher level software such as Simulink or a custom ROS node.

The source code for OpenCR can be downloaded by cloning the following repository.

`https://github.com/hanyiabc/ACSL_turtlebot3.git`

Arduino IDE can be used to burn the **turtlebot3_core.ino** file located in **ACSL_turtlebot3/src/turtlebot3_core** directory. Instructions for setting up the Arduino IDE for `TurtleBot3` can be found in the following link.

`https://emanual.robotis.com/docs/en/parts/controller/opencr10/#arduino-ide`

2 Syncing Time using NTP Server and Client

Raspberry Pi and Odroid SBCs do not have a hardware clock, and therefore it relies on an active internet connection to fetch the accurate time from Ubuntu time server. However, this is not possible as Linksys router, does not have internet access.
Every time SBC boots up, the time has to be updated. The time on the Remote PC and the SBC have to be synced up in order for the various transformations to work for the purpose of running SLAM and Navigation in the ACSL Lab Environment. Hence, to sync up time between these two machines, NTP protocol is being used.
NTP or Network Time Protocol is a protocol that is used to synchronize all system clocks in a network to use the same time. When we use the term NTP, we are referring to the protocol itself and also the client and server programs running on the networked computers. NTP belongs to the traditional TCP/IP protocol suite and can easily be classified as one of its oldest parts.
When you are initially setting up the clock, it takes six exchanges within 5 to 10 minutes before the clock is set up. Once the clocks in a network are synchronized, the client(s) update their clocks with the server once every 10 minutes. This is usually done through a single exchange of message(transaction). These transactions use port number 123 of your system.
In this section, a step-by-step procedure is described on how to:

- Install and configure the NTP server on a Ubuntu machine.
- Configure the NTP Client to be time synced with the server.

These instructions are taken from `https://vitux.com/how-to-install-ntp-server-and-client-on-ubuntu/`. However, it is advised that the following instructions are followed as there are some project specific changes that have been made.

2.1 Configure NTP Server on the Remote PC

Open Terminal and run the following commands **on the remote PC**

```
sudo apt-get update
sudo apt-get install ntp
```

Most likely NTP will already be installed on the machine. After installation, the configuration file needs to be altered and server pools closest to the PC location needs to be added. This step is optional, as most likely the servers will be configured correctly.

```
sudo gedit /etc/ntp.conf
```

Replace/edit the server pools with the following.

```
pool 0.us.pool.ntp.org iburst
pool 1.us.pool.ntp.org iburst
pool 2.us.pool.ntp.org iburst
pool 3.us.pool.ntp.org iburst
```

Save and close the configuration file. In order for the above changes to take effect, you need to restart the NTP server. The time from these servers will be synced with the time on the remote PC. as it has internet access. This will make sure that the NTP server clock is accurate.

```
sudo service ntp restart
```

Run the following command to check if the NTP server is configured properly. You should see the remote PC connecting to various servers.

```
sudo service ntp status
```

Finally, the system's UFW firewall needs to be configured so that incoming connections can access the NTP server at UDP Port number 123. This is the most important step, as it will configure the remote PC to allow client connections.

```
sudo ufw allow from any to any port 123 proto udp
```

Now the Ubuntu host machine is configured to be used as an NTP server. Please take a note of the IP address of the remote PC when it is connected to the appropriate router. IP address of a machine depends on which router it is connected to, and hence is not constant. We will need this address in the next step.

2.2 Configure NTP Client on the SBC

The `ntpdate` command will let you manually check your connection configuration with the NTP-server. Only in this case, the server would be the one that we specify. It should already be installed on the SBC, but in case it is not, run the following command **on the SBC**.

```
sudo apt-get install ntpdate
```

As the SBC will be a client to the host we set up on the remote PC, we need to modify the hosts file for NTP.

```
sudo gedit /etc/hosts
```

Now add your NTP server's IP (remote PC IP address noted in the previous step) and specify a hostname. Assuming that the remote PC IP address is `192.168.1.183` and the server has to be named `ACSLServer`, add the following line to this file:

```
192.168.1.183          ACSLServer
```

Save and close the NTP hosts file. Because we want our client to sync time with the NTP server, the `timesyncd` service on the client machine needs to be disabled by running the following command.

```
sudo timedatectl set-ntp off
```

Now, we need to specify this NTP server on the remote PC in the NTP configuration file of the client. If `NTP` is not installed install it using the following command.

```
sudo apt-get install ntp
```

Now we want our client machine to use our own NTP host server to be used as the default time server. For this, we need to edit the `/etc/ntp.conf` file on the client machine.

```
sudo nano /etc/ntp.conf
```

Add the following line to the file before the list of other servers. There should be a comment indicating the location where new servers can be added, although this is only for readability and functionality shouldn't be affected.

```
server ACSLServer prefer iburst
```

Finally restart the `NTP` service and the time should be synced.

```
sudo service ntp restart
```

Now your client and server machines are configured to be time-synced. You can view the time synchronization queue by running the following command:

```
ntpq -p
```

If `ACSLServer` is seen in the list of servers then the client is properly configured.
Finally the timezone needs to be set on the SBC to `America/New-York`. On the Odroid XU4 running `Ubuntu` with `MATE` desktop, it is a simple command line instruction as follows.

```
sudo timedatectl set-timezone America/New-York
```

This command may not work on the Raspberry Pi 3B+, but the timezone can be changed using the `raspi-config` utility.
Once the timezones are set, the SBCs are configured properly.

3 Bring up the Turtlebot

Once the remote PC and Raspberry PI are setup according to the instructions provieded in the previous sections, Follow the instructions below to bring up turtlebot.
On Remote PC

```
roscore
```

On SBC

```
roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

On Remote PC, once you followed the instructions for building the workspace and setting up the environment variables, load the environment variables related to this workspace (Note: This has to be done everytime you want to run something from this workspace from a new termianl)

```
cd ASCL_turtlebot3/
source devel/setup.bash
```

To run teleop to control the Turtlebot with keyboard.

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

To run the navigation with PID velocity effort controller and SLAM (with localizatin provided by SLAM and no initial map required) on the physical robot.

```
roslaunch turtlebot3_bringup turtlebot3_physical_nav_control.launch
```

To run the navigation with PID velocity effort controller and SLAM (with localizatin provided by SLAM and no initial map required) on Gazebo for simulation.

```
roslaunch turtlebot3_bringup turtlebot3_sim_nav_control.launch
```

To run SLAM only to draw maps Choose a pre-defined Gazebo world from the following:

– turtlebot3_empty_world.launch

– turtlebot3_world.launch

– turtlebot3_house.launch

– turtlebot3_stage_1.launch

– turtlebot3_stage_2.launch

– turtlebot3_stage_3.launch

– turtlebot3_stage_4.launch

For example, if **turtlebot3_world.launch** is chosen, run the following command to bring up a virtual Turtlebot on Gazebo for simulation

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Then, run the following commands to launch the SLAM nodes and remote control using keyboard.

```
roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Once you are satisfied with the map, save the map by running

```
rosrun map_server map_saver -f ~/map
```

4 Repository Introduction

The repository is a catkin workspace. A catkin workspace has a src folder that contains multiple packages. The worth-mentio ning ones are:

- hls_lfcd_lds_driver
- raspicam_node
- ros_control
- simulink
- turtlebot3
  - turtlebot3_bringup
  - turtlebot3_navigation
  - turtlebot3_slam
  - turtlebot3_teleop
- turtlebot3_core
- turtlebot3_setup_motor
- turtlebot3_simulations

Not all packages are catkin package. **simulink**, **turtlebot3_core turtlebot3_setup_motor** are all just directories. All the rests are properly configured as catkin packages, meaning all the nodes defined under the package will be compiled when running the **catkain_make** command.

hls_lfcd_lds_driver and **raspicam_node** are drivers for the PI Camera and lidar. These will be running in the single board computer.

ros_control contains all the configuration and nodes necessary to control the robot. It contains the PID controller configurationl, PID gains, etc. Two launch files are included for controlling the robot in either the simulations or in the physical world. They are:

- turtlebot3_control.launch
- turtlebot3_control_simulation.launch

These launch file are not ran directly, instead they are inlcuded by other launch file that we will introduce later. The PID gains can be tuned by either changing the parameters in the launch file or use a ros package called rqt_reconfiture to adjust parameters with sliders at runtime. There are 2 nodes under the **ros_control** package. They are **differential_driver.py** and **ros_control_node**. The **differential_driver.py** is a Python ros node that split the incoming **/cmd_vel** messages into left and right velocity for the differential drive robot based on vehicle geometry. The **cmd_vel** is the standard topic that ROS nav stack used to output the velocity command. It commands the robot in terms of translational velocity and orientation. The **ros_control_node** is a C++ ros node compiled from the soruce file: **message_redirect.cpp**. This node simply takes the left and right wheel angular velocities feedbacks from joint_states converts them to linear velocities and publish them into 2 Float64 messages for the PID controller to read.

The directory **simulink** contains all the MATLAB scrips and Simulink models for controlling the robot with MATLAB

turtlebot3_bringup contains all the launch file for bringing up the Turtlebot in the SBC and the remote PC. **turtlebot3_physical_nav_control.launch** was added to the bringup package to provide a convinient one-file launch that will handles everything. This launch file brings up the turtlebot on the remote PC, runs the velocity control and the navigation stack with SLAM. A simulation version will be provided in the future.

turtlebot3_description contains the URDF and Gazebo descripion of the turtlebot for both simulation and physical robot.

For Waffle Pi, **turtlebot3_waffle_pi.gazebo.xacro** and **turtlebot3_waffle_pi.urdf.xacro** are provided by the turtlebot package, however, for effort control, we added 2 more files. They are: **turtlebot3_waffle_pi_effort_controller.gazebo.xacro** and **turtlebot3_waffle_pi_effort_controller.urdf.xacro** These files are derived from the original descriptions and added ability to control effort in both the simulation and the physical robot.

turtlebot3_navigation contains the default configuration for running the ROS navigation stack with the Turtlebot. A new launch file **turtlebot3_navigation_no_map.launch** was created. This launch file derived from the original navigation launch file, but the difference is that this configuration doesn't need a SLAM map for localizatin. The localizatin is provide by SLAM instaed of Adaptive Monte Carlo Localizatin. The default launch file **turtlebot3_navigation.launch** will run navigation assuming that a SLAM map has been drawn by running SLAM before. In this configuration, localizatin is provided by AMCL.

5 Torque Control using Dynamixel

Dynamixel is a microcontroller based actuator with in-built PID controller. OpenCR communicates with Dynamixel using packet communication via TTL/RS45 ports.

Hardware level abstraction is achieved via Dynamixel SDK library available in several programming languages. (CPP used in our case.) Dynamixel register addresses and byte sizes for both RAM and EEPROM memory provided in control table. RAM is most frequently used memory for robot applications, while startup settings are stored on EEPROM.

The Control Table is a structure that consists of multiple Data fields to store status or to control the device. Users can check current status of the device by reading a specific Data from the Control Table with Read Instruction Packets. WRITE Instruction Packets enable users to control the device by changing specific Data in the Control Table. Packet sizes range from 1 – 4 bytes.

Following is a snapshot of the Dynamixel EEPROM Control Table. Each data in the Control Table is restored to initial values when the device is turned on. Default values in the EEPROM area (addresses 0-63) are initial values of the device (factory default settings). If any values in the EEPROM area are modified by a user, modified values will be restored as initial values when the device is turned on. Initial Values in the RAM area are restored when the device is turned on.

Address	Size (Byte)	Data Name	Access	Default Value	Range	Unit
0	2	Model Number	R	1,030	-	-
2	4	Model Information	R	-	-	-
6	1	Firmware Version	R	-	-	-
7	1	ID	RW	1	0 ~ 253	-
8	1	Baud Rate	RW	1	0 ~ 7	-
9	1	Return Delay Time	RW	250	0 ~ 254	2 [μsec]
10	1	Drive Mode	RW	0	0 ~ 1	-
11	1	Operating Mode	RW	3	0 ~ 16	-
12	1	Secondary(Shadow) ID	RW	255	0 ~ 252	-
13	1	Protocol Type	RW	2	1 ~ 2	-
20	4	Homing Offset	RW	0	-1,044,479 ~ 1,044,479	1 [pulse]
24	4	Moving Threshold	RW	10	0 ~ 1,023	0.229 [rev/min]
31	1	Temperature Limit	RW	80	0 ~ 100	1 [°C]
32	2	Max Voltage Limit	RW	160	95 ~ 160	0.1 [V]
34	2	Min Voltage Limit	RW	95	95 ~ 160	0.1 [V]
36	2	PWM Limit	RW	885	0 ~ 885	0.113 [%]
38	2	Current Limit	RW	1,193	0 ~ 1,193	2.69 [mA]
44	4	Velocity Limit	RW	330	0 ~ 1,023	0.229 [rev/min]

For the purpose of torque control, the operating mode of the dynamixel motor needs to be set to **MODE 0**. This can be done using the motor setup code in the git repository. To change the operating mode to **MODE 0**, flash the turtlebot3_setup_motor firmware to the OpenCR board while the motors are connected to the board. Open up a serial terminal with Arduino or Terraterm, under the serial termianl, you will see the following options:

```
    1. setup left  motor
    2. setup right motor
    3. test  left  motor
    4. test  right motor
    5. setup left  motor for current control
    6. setup right  motor for current control
    7. test left  motor for current control
    8. test right  motor for current control
>>
```

The first 4 options comes with the OpenCR Arduino library. The last 4 options are added based on the first 4 options and they configure the left and right motors for torque control. For now, the current is limited to half of the highest value to avoid damage to the hardware. If changes are needed, the source code needs to be modified and the motors has to be setup again. To setup the left motor, simply put 5 in the termianl and press enter. Wait until it says it's finished. Then use option 6 for the right motor. After the setup is done, use option 7 and 8 to test the motor. The test option apply a small torque to the motor for one second and stops for one second. Make sure both motors are tested then, flash the turtlebot3_core firmware back. Once the operating mode is set. The OpenCR code is changed to write torque values to the Dynamixel RAM addresses, when the robot is live. The addresses that are useful for this purpose are given in the image below.

102	2	Goal Current	RW	-	-Current Limit(38) ~ Current Limit(38)	2.69 [mA]
104	4	Goal Velocity	RW	-	-Velocity Limit(44) ~ Velocity Limit(44)	0.229 [rev/min]
108	4	Profile Acceleration	RW	0	0 ~ 32,767 0 ~ 32,737	214.577 [rev/min^2] 1 [ms]
112	4	Profile Velocity	RW	0	0 ~ 32,767	0.229 [rev/min]
116	4	Goal Position	RW	-	Min Position Limit(52) ~ Max Position Limit(48)	1 [pulse]
120	2	Realtime Tick	R	-	0 ~ 32,767	1 [msec]
122	1	Moving	R	0	0 ~ 1	-
123	1	Moving Status	R	0	-	-
124	2	Present PWM	R	-	-	-
126	2	Present Current	R	-	-	2.69 [mA]
128	4	Present Velocity	R	-	-	0.229 [rev/min]
132	4	Present Position	R	-	-	1 [pulse]

The code that change the operating mode of the motor is below.

```
bool setupMotorLeftCurrentControl(void)
{
  CMD_SERIAL.println("Setup Motor Left for current control...");


  if (tb3_id < 0)
  {
    CMD_SERIAL.println("  no dxl motors");
  }
  else
  {
    write(portHandler, packetHandler2, tb3_id, 64, 1, 0);
    write(portHandler, packetHandler2, tb3_id, 7, 1, 1);
    tb3_id = 1;
    write(portHandler, packetHandler2, tb3_id, 8, 1, 3);
    portHandler->setBaudRate(1000000);
    write(portHandler, packetHandler2, tb3_id, 10, 1, 0);
    write(portHandler, packetHandler2, tb3_id, 11, 1, 0);

    // this set the limit to half
    write(portHandler, packetHandler2, tb3_id, 38, 2, 596);
    CMD_SERIAL.println("  Warning: Current limit is set to half to protect the motor!");
    CMD_SERIAL.println("  ok");
```

```
  }
}

bool setupMotorRightCurrentControl(void)
{
  CMD_SERIAL.println("Setup Motor Right for current control...");


  if (tb3_id < 0)
  {
    CMD_SERIAL.println("  no dxl motors");
  }
  else
  {
    write(portHandler, packetHandler2, tb3_id, 64, 1, 0);
    write(portHandler, packetHandler2, tb3_id, 7, 1, 2);
    tb3_id = 2;
    write(portHandler, packetHandler2, tb3_id, 8, 1, 3);
    portHandler->setBaudRate(1000000);
    write(portHandler, packetHandler2, tb3_id, 10, 1, 1);
    write(portHandler, packetHandler2, tb3_id, 11, 1, 0);

     // this set the limit to half
    write(portHandler, packetHandler2, tb3_id, 38, 2, 596);
    CMD_SERIAL.println("  Warning: Current limit is set to half to protect the motor!");
    CMD_SERIAL.println("  ok");
  }
}
```

The code used to test the torque control after setup is below.

```
void testMotorCurrentControl(uint8_t id)
{
  uint32_t pre_time;
  uint8_t toggle = 0;

  if (id == 1)
  {
    CMD_SERIAL.printf("Test Motor Left...");
  }
  else
  {
    CMD_SERIAL.printf("Test Motor Right...");
  }
  // We run at 1000000
  portHandler->setBaudRate(1000000);

  uint16_t model_number;
  int dxl_comm_result = packetHandler2->ping(portHandler, id, &model_number);
  if (dxl_comm_result == COMM_SUCCESS)
  {
    CMD_SERIAL.printf(" found type: %d\n", model_number);
    write(portHandler, packetHandler2, id, 64, 1, 1);

    toggle = 0;
    pre_time = millis();
    write(portHandler, packetHandler2, id, 102, 2, 100);
    while (1)
    {
```

```
      if (CMD_SERIAL.available())
      {
        flushCmd();
        break;
      }

      if (millis() - pre_time > 1000)
      {
        pre_time = millis();

        toggle ^= 1;

        if (toggle)
        {
          write(portHandler, packetHandler2, id, 102, 2, 0);
        }
        else
        {
          write(portHandler, packetHandler2, id, 102, 2, 100);
        }
      }
    }
    write(portHandler, packetHandler2, id, 102, 2, 0);
  }
  else
  {
    CMD_SERIAL.printf("  dxl motor ID:%d not found\n", id);
  }
}
```

The OpenCR Arduino library comes with a TurtleBot3MotorDriver class that helps controlling the 2 motors by commanding velocities. A new class TurtleBot3MotorTorqueDriver was derived (strictly speaking, not inherited, instead just copied and pasted) from the TurtleBot3MotorDriver with new member functions added for controlling the torque instead of velocities. The class is defined as below

```
class TurtleBot3MotorTorqueDriver
{
public:
    TurtleBot3MotorTorqueDriver();
    ~TurtleBot3MotorTorqueDriver();
    bool init(String turtlebot3);
    void close(void);
    bool setTorque(bool onoff);
    bool getTorque();
    bool readEncoder(int32_t &left_value, int32_t &right_value);
    bool writeVelocity(int64_t left_value, int64_t right_value);

    bool writeTorque(int16_t left_value, int16_t right_value);
    bool readTorque(float &left_torque, float &right_torque);
    bool controlMotor(const float wheel_radius, const float wheel_separation, float *value);
    bool controlMotor(float *torque);


private:
    uint32_t baudrate_;
    float protocol_version_;
    uint8_t left_wheel_id_;
```

```
    uint8_t right_wheel_id_;
    bool torque_;

    uint16_t dynamixel_limit_max_velocity_;
    uint16_t dynamixel_limit_max_current_;

    dynamixel::PortHandler *portHandler_;
    dynamixel::PacketHandler *packetHandler_;

    dynamixel::GroupSyncWrite *groupSyncWriteVelocity_;
    dynamixel::GroupSyncRead *groupSyncReadEncoder_;

    dynamixel::GroupSyncWrite *groupSyncWriteTorque_;
    dynamixel::GroupSyncRead *groupSyncReadTorque_;
```

The following member functions is added to the TurtleBot3MotorDriver class, which takes in input torque value and converts it into appropriate register values for writing to the appropriate addresses.

```
bool TurtleBot3MotorTorqueDriver::controlMotor(float *torque)
{
       bool dxl_comm_result = false;

       int16_t wheel_current_cmd[2];
       wheel_current_cmd[LEFT] = CURRENT_TO_OUTPUT(TORQUE_TO_CURRENT(torque[LEFT]));
       wheel_current_cmd[RIGHT] = CURRENT_TO_OUTPUT(TORQUE_TO_CURRENT(torque[RIGHT]));

       wheel_current_cmd[LEFT] = constrain(wheel_current_cmd[LEFT], -dynamixel_limit_max_current_,
           dynamixel_limit_max_current_);
       wheel_current_cmd[RIGHT] = constrain(wheel_current_cmd[RIGHT],
           -dynamixel_limit_max_current_, dynamixel_limit_max_current_);

       dxl_comm_result = writeTorque((int16_t)wheel_current_cmd[LEFT],
           (int16_t)wheel_current_cmd[RIGHT]);

       if (dxl_comm_result == false)
          return false;

       return true;
```

CURRENT_TO_OUTPUT and **TORQUE_TO_CURRENT** are macro functions that does the conversions from torque to current and current to a 2 byte value that the microcontroller undertand. All the necessary macros are defined as

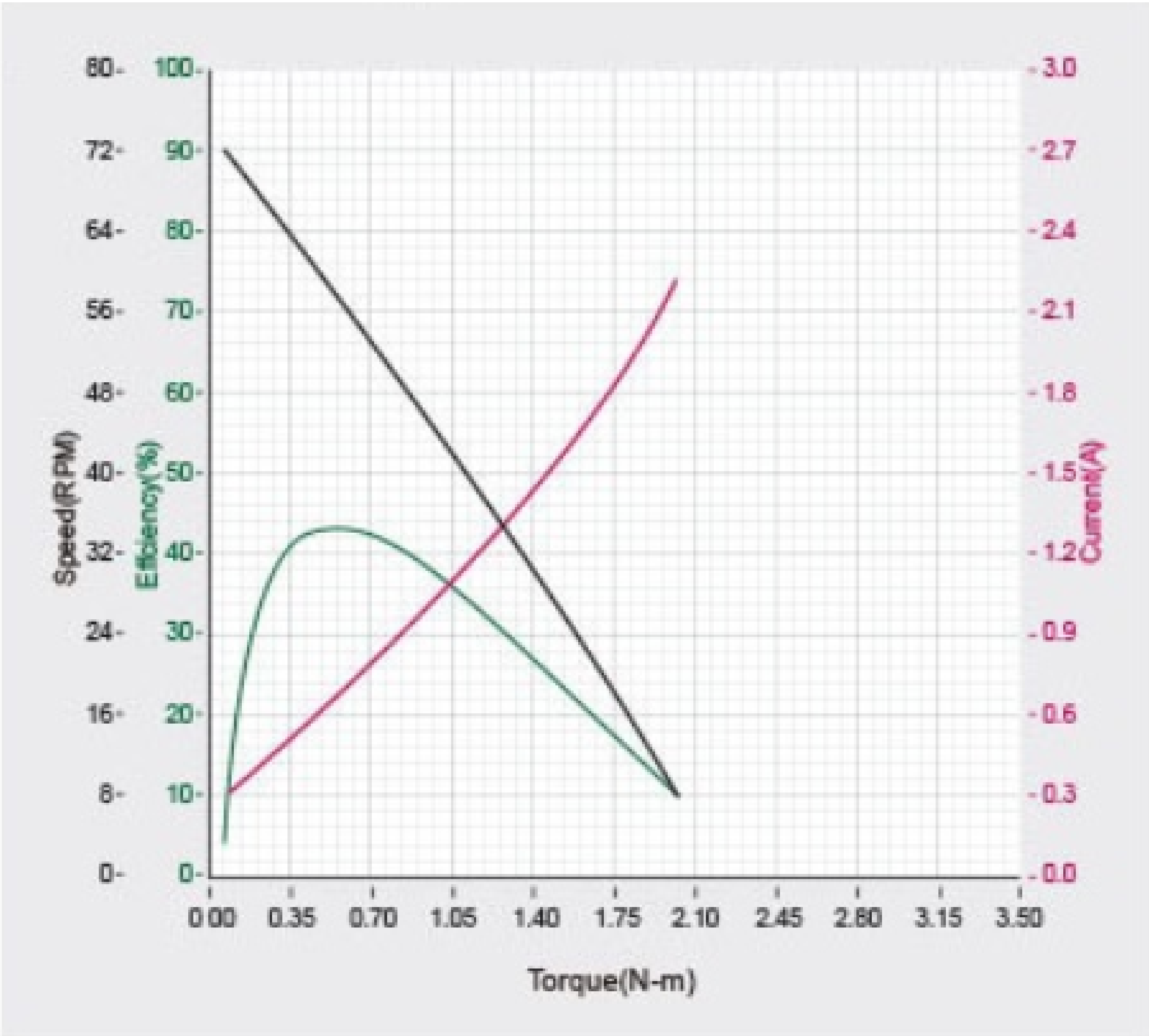
```
#include <stdint.h>
#define WAFFLE_DXL_LIMIT_MAX_CURRENT 780
#define ADDR_X_GOAL_CURRENT        102
#define ADDR_X_PRESENT_CURRENT     126
#define LEN_X_GOAL_CURRENT         2
#define LEN_X_PRESENT_CURRENT      2

#define MAX_CURRENT_11V1             2.1
#define MAX_TORQUE_11V1              2.7

#define CURRENT_GOAL_UNIT            2.69
#define TORQUE_TO_CURRENT(t)         t * (MAX_CURRENT_11V1 / MAX_TORQUE_11V1)// convert torque to
    current in amp
#define CURRENT_TO_OUTPUT(a)         (int16_t)(a * 1000 / CURRENT_GOAL_UNIT)
#define CURRENT_TO_TORQUE(t)     t / (MAX_CURRENT_11V1 / MAX_TORQUE_11V1)// convert current in amp
```

```
    to torque in N-m
```

This conversion is based on a linear approximation of the actuator performance graph.



This function below takes the converted value and write to both motors using the Dynamixel SDK APIs.

```
bool TurtleBot3MotorTorqueDriver::writeTorque(int16_t left_value, int16_t right_value)
{
    bool dxl_addparam_result;
    int8_t dxl_comm_result;

    dxl_addparam_result = groupSyncWriteTorque_->addParam(left_wheel_id_, (uint8_t *)&left_value);
    if (dxl_addparam_result != true)
        return false;

    dxl_addparam_result = groupSyncWriteTorque_->addParam(right_wheel_id_, (uint8_t
        *)&right_value);
    if (dxl_addparam_result != true)
        return false;
```

```
    dxl_comm_result = groupSyncWriteTorque_->txPacket();
    if (dxl_comm_result != COMM_SUCCESS)
    {
        Serial.println(packetHandler_->getTxRxResult(dxl_comm_result));
        return false;
    }

    groupSyncWriteTorque_->clearParam();
    return true;
}
```

Subscribers are added to subscribe to the topics left_torque right_torque. The following use the function defined above and write the correspounding values to the correct memory location of the Dynamixel motor controller when new message comes in. When there is no new message, it will timeout and write zeros to both motors.

```
  if ((t-tTime[0]) >= (1000 / CONTROL_MOTOR_TORQUE_FREQUENCY))
  {

    updateGoalTorque();
    //this timeout will stop the motor if no message comes in

    if ((t-tTime[6]) > CONTROL_MOTOR_TIMEOUT)
    {
      motor_driver.controlMotor(zero_torque);
    }
    else {
      motor_driver.controlMotor(torque);
    }
```

The Odometry publisher is in the microcontroller. Since the built-in differential drive plugin which provides Odometry was removed to use our own controller, An Odometry publisher was added to the system. The source code is below.

```
  if ((t-tTime[0]) >= (1000 / CONTROL_MOTOR_TORQUE_FREQUENCY))
  {

    updateGoalTorque();
    //this timeout will stop the motor if no message comes in

    if ((t-tTime[6]) > CONTROL_MOTOR_TIMEOUT)
    {
      motor_driver.controlMotor(zero_torque);
    }
    else {
      motor_driver.controlMotor(torque);
    }
```

6 Turtlebot Graphical User Interface

This section documents the graphical user interface for manipulating the TurtleBot3 in both real world and simulated environments. It contains descriptions for each graphical widget and code documentation. The turtlegui package contains all of the c++ scripts and dependency files responsible for the interface. And once the package is appropriately sourced via the terminal, it can be run with the following command:

```
rosrun turtlegui turtlegui
```

6.1 Qt

The interface runs on Qt, a cross-platform graphical toolkit. This module implements a variety of important classes, the first of which is QApplication. The source code for the instantiation of the Turtlebot qt application, in main.cpp, is

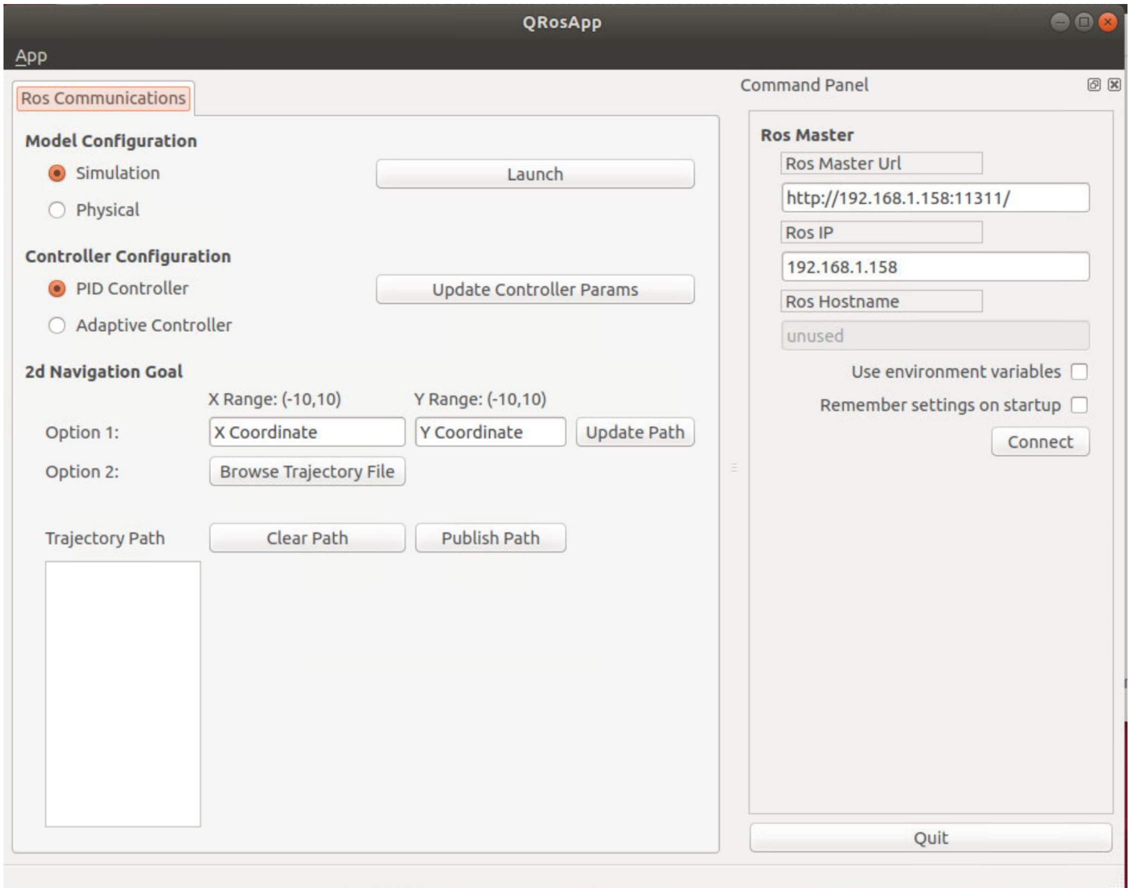
The QApplication class takes care of a lot of things, most notably the input CLI arguments and the event loop. The event loop is a loop that waits for user input in GUI applications, and is launched on this command

Another important Qt class is QWidget, which is used to describe the widgets in the interface. Widgets are the basic building blocks for a GUI application, and are able to respond to user events. All of the graphical elements in the interface inherit from the QWidget class, including QPushButton, QRadioButton, and QLineEdit. Widgets are also responsible for the handling of the signal and slot mechanism. On a high level, signals and slots are used for communication between objects. In GUI programming, when one widget changes, other widgets need to be notified of this change. Rather than hard-wiring the code that generates these events, signals and slots allow for fluid communication between objects.

As seen in the line of code below, the lastWindowClosed() signal connects to the quit() slot so that when the exit button is clicked in the GUI, the program terminates. The Qt toolkit is a very useful module for handling the front-end side of programming and allows for a seamless transition to back-end code.

6.2 GUI Widget Walkthrough

This section gives a detailed description for each widget in main window of the interface. A screenshot of the interface can be seen below



On startup, a ROS Master is initialized, this provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer. All ROS commands are handled implicitly, thus requiring a single terminal window.

The three main sections of the GUI are Model Configuration, Controller Configuration, and 2d Navigation Goal.

6.2.1 Model Configuration

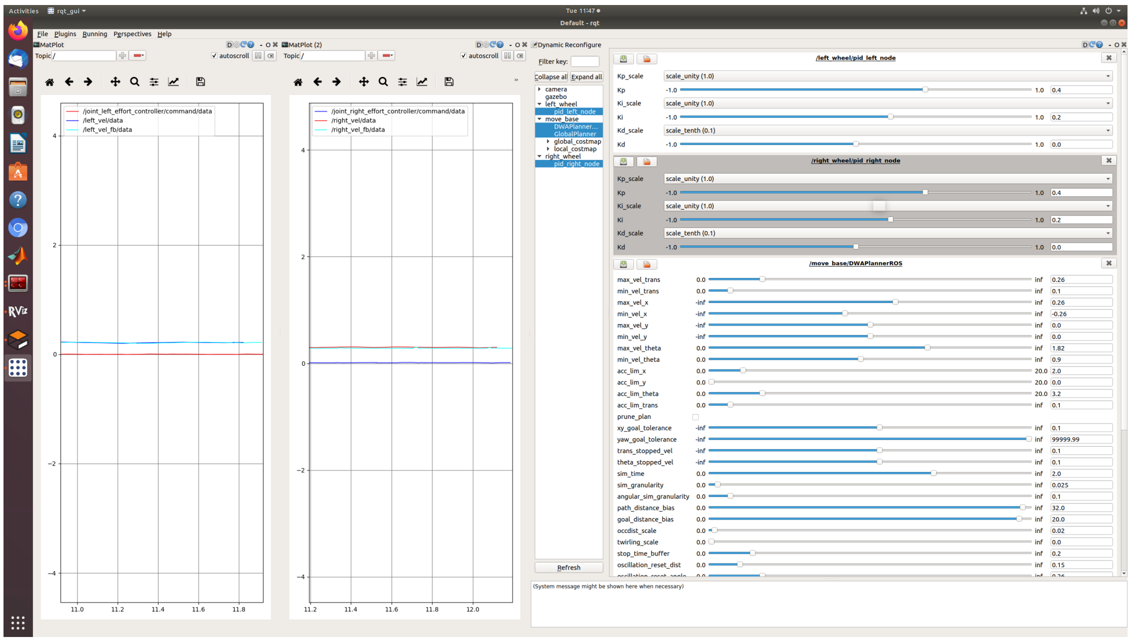
The first tab contains two QRadiobuttons and one QPushButton that allow the user to specify and launch an environment for the Turtlebot. The simulation launches Gazebo and RVIZ instances for visualization. This is accomplished by running this ROS command: The ROS launch command is blocking, which means once it is executed, another terminal window must be opened to run any additional commands. The ampersand at the end of the command tells the process to run in the background, allowing for more commands to be run in the same window. The real-world option uses the same syntax but launches a different package:

Once an environment is loaded, never before, the dynamics and path of the turtlebot can be manipulated as seen in the following sections. Since there are multiple ROS processes running in the same terminal, ROS has trouble shutting them all down properly and tends to stall the terminal. In order to expedite the shutdown, the timeout time specified in following file was modified:

```
/opt/ros/melodic/lib/python2.7/dist-packages/roslaunch/nodeprocess.py
```

6.2.2 Controller Configuration

This section allows the user to manipulate the parameters of the turtlebot. Again it contains two QRadiobuttons and one QPushButton for configuring the controller for the Turtlebot. It is important to note that only the PID controller option has been implemented, the Adaptive Controller does not do anything. The update controller parameters button runs on rqt, a software framework of ROS that implements the various GUI tools in the form of plugins. More specifically it uses an rqt metapackage called rqt_gui, which allows multiple rqt plugins to be docked in a single window. The command that handles the instantiation and reference to this framework is: This launches a new window for viewing all ROS nodes and topics, configuring the PID parameters, and graphically analyzing the error of the PID controller. A screenshot of the window is shown below



6.2.3 2d Navigation Goal

The turtlebot is able to follow one or more precise waypoints specified by the user via the 2D Navigation Goal section of the interface. Option 1 allows for the publishing of a single waypoint while option 2 lets the user browse for a csv file containing a list of waypoints. The trajectory path inherits the QStringListModel class that keeps track of the robot's current path. Once either option is chosen, the trajectory path will be updated. The clear_path button simply removes all items from the trajectory path. To get the Turtlebot to actually start moving and following the path, the publish_path button must be pressed.

This button utilizes the follow_waypoints package, which handles the low-level robot communication for following a path. Three critical ROS commands are executed, the first of which can be seen below.

This creates three ROS topics, /initialpose, /path_ready, and /path_reset. The /path_reset topic is published when the clear path button is clicked. The input trajectory path is published to the /initialpose topic. This topic is of type geometry_msgs/PoseWithCovarianceStamped, thus the messages published to this topic have to be of this type. This message type contains two fields, std_msgs/Header header and geometry_msgs/PoseWithCovariance pose. The former field is just used for communicate timestamped data in a particular coordinate frame. The latter has two more implicit fields, geometry_msgs/Pose pose and float64[36] covariance. The pose field represents a pose in free space with uncertainty. The covariance field is a row-major representation of the 6x6 covariance matrix, which specifies the uncertainty of the pose field. The pose field has two more implicit fields, geometry_msgs/Point position and geometry_msgs/Quaternion orientation. The former contains the position of a point in free space, with fields: float64 x, float64 y, float64 z. The latter represents an orientation in free space in quaternion form, with fields float64 x, float64 y, float64 z, float64 w. By default, the covariance, orientation, and pose.z are all zero, thus the user only needs to specify an x and y position. It is important to note that if the specified goal is not close to current position of the Turtlebot, it will not move. The trajectory.csv file can be used to get an idea of the coordinate pairs the Turtlebot will be able to move to.

Once the path is published, another ROS command is executed: This just publishes an empty message to /path_ready which tells the Turtlebot to start following the path published to /initialpose. Once the Turtlebot reaches the last point in the path, it will stop moving and a new path will have to be specified. The Turtlebot always starts at the origin, and its position is continuously published to the /odom ROS topic.